END
DATE
FILMED

MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS

AD-A174 874

RSRE MEMORANDUM No. 3947

RSRE
MEMORANDUM No. 3947

ROYAL SIGNALS & RADAR ESTABLISHMENT

REMOTE CAPABILITIES IN COMPUTER NETWORKS

Authors: J M Foster, I F Currie

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3947

Title: Remote Capabilities in computer networks

Authors: J.M.Foster and I.F.Currie

Date: March 1986

Summary

System-wide types, capabilities and procedure values can be used
in computer networks to improve control of access and flexibility.
This paper describes a network system based on these ideas. They
provide a standard, uniform way of referring to data and using
procedures in remote computers, with control over access and system
type-checking. A working experimental version of the system exists,
implemented on *Flex computers, which provide microcoded support of*
capabilities. The mechanisms are such that normal programs written
in terms of procedures, even those using procedure parameters, can
be converted into network services without change. A possible way of
implementing a similar scheme on conventional hardware is discussed.

1 Introduction

Capabilities are unforgeable values which programs can manipulate
(Fabry 1974, England 1975, Needham and Walker 1977). For each kind
of capability some operations are permitted which are forbidden on
other kinds of capability and on non-capabilities. A particular
capability might give read access to an area of store. Programs will
be able to read that store only if they have, or can obtain, the
capability value. Other capabilities might give the ability to address
a particular piece of backing store, to use a peripheral or call a
procedure. In order to guarantee unforgeability, the checking of
capabilities is usually performed by microcode. The earlier
implementations of capabilities worked by segregating them in
special areas, but there is much to be gained by making them into
values which are freely mobile. If this is done they must still be
unforgeable, which can be ensured by using tag bits under the sole
control of the microcode. This also makes it easy and efficient to use
the notions of capability at a fine-grained level. Flex (Foster et al.
1979, Currie, Edwards and Foster 1981) is a computer architecture of
this kind.

We can extend the idea of capabilities to networks of computers by
allowing one computer to hold a capability for something inside
another. These capabilities can be exercised across a network. They
must still be unforgeable, both within the machines and in transit
across the network. We shall refer to them as remote capabilities.

Capabilities enable us to control access. It is a particular virtue
that this control is provided once and for all in the microcode, and
all the protection derives from this microcode. So if we want to
verify or validate or learn to trust the access control mechanisms,
we have only to examine the microcode. It is true that this may not be
a trivial task, but it is a limited one. Extending these ideas to
networks means that we have only to examine the remote capability
mechanism, and we have provided a way of controlling access across
the network.

If these mechanisms are built into the machine in the microcode
by tagging, then we have control which cannot be vitiated by wrong
compilers or wrong assembly code. The protection is provided at the
level of machine words. Each capability consists of a word, with its
associated tag bit which is accessible only to the microcode. The
rules controlling the use of this word cannot be violated. But the
microcode attaches no meaning to groups of words. Any relations
between different capabilities and non-capabilities within the same
structure or record are unknown to it.

1

The type system of most programming languages provides a means for describing data structures which are composed of many words and may be distributed around the store. This provides flexibility and convenience, in that many operations can be described more concisely and programs become clearer. Furthermore it also provides a sort of access control: for example, the checking mechanisms may prevent integers being used as addresses. It has been suggested that this mechanism is sufficient to provide the needed control of access and capability mechanisms might be reserved for grosser checks, thus allowing a cruder, easier and less fine-grained implementation. However, the language types are normally only meaningful inside an individual program. There may be communication of type information between different modules of a program, but data can get from one program into a separate one only through the operating system. This knows nothing of the types within those programs, so type checking between programs is lost. Thus holds still more stringly if the programs are in different languages. For some time the operating system of the Flex computers has used its own types to overcome this omission (Currie and Foster 1982). The Flex system types are sufficient to describe all the data structures produced by Algol68, Ada, Pascal, ML and other languages, together with types appropriate to the description of backing store values. Values produced by these languages can be passed between separate programs, which might be written in different languages, and type checking is maintained. Operating system procedures can be created using the languages, in such a way as to produce typed procedures which can be invoked from a command interpreter, and again type checking is performed.

The correctness of these types depends on the correctness of the compilers, unlike the capability control provided by the microcode. The Flex types have to be independent of programs and of the precise form of the text defining the programs. Though the Flex types describe the data structures produced by Ada and other languages, this is not because they are a superset of all kinds of type definition, but because every data structure produced by the languages has a description in terms of Flex types. These types have now been extended to network communication, providing network-wide types and type checking. They have to be independent not merely of particular programs but also of computers; indeed the types must be understood throughout the network. This is particularly important for new abstract types, which may be invented on one machine, but must be understood and checked by all the computers on the network.

There are thus two levels of access control in the system which is being described: capabilities, which work on individual words and depend only on the correctness of the microcode, and types, which

work on groups of words but depend on the correctness of the compilers.

Network-wide types not only enable us to perform checks, but also function as a description of data. This function of types is used in the Courier protocol (Xerox Corporation 1981), but the types we are about to describe are very much more powerful. It will be seen that we can transfer capabilities for dynamically created procedures around the network, and thus send new procedure capabilities to other machines to be called in due course. The types of the procedures, which of course contain the types of their parameters (including procedure parameters), describe how their parameters are to be handled. They give in an abbreviated form a description of what can be done with them and make explicit the possible protocols.

Procedures are free-standing values in Flex (Foster, Currie and Edwards 1982). The values and references that they need to use are bound into them. A capability for a procedure can be created so that it provides the only access in the whole system to some particular object. This procedure can then give arbitrarily programmed control over access to that object. These procedures, which are proper values, allow an "object-oriented" approach to the creation of new abstract data types, in that packages of procedure values implement the possible operations on an abstract type. Flex permits procedures or packages of procedures to be written to the backing stores - a Flex computer can have several backing stores - and so new abstract data types and their values can be written to backing store, with the appropriate type checking on their use. Such procedure values can have an indefinitely long life.

Flex uses a remote procedure call (White 1976) mechanism for its network. On Flex, unlike other RPC networks, the possible procedures which can be called do not have to be agreed between the machines from the start. New remote procedure values can be made up by any program, capabilities for using them can be passed to another computer and then used for remote calls. The mechanisms allow for exception handling in remote calls, of which timeout is one example, and they allow for aborting or breaking in to remote calls.

The discussion in this paper is independent of the failure semantics of the network. Flex operates an "at least once" semantics (Spector 1982, Liskov and Scheifler, 1983) but all the following remarks would apply equally to "exactly once" or to "at most once" semantics.

Capabilities encourage the use of values rather then names, and Flex takes full advantage of this throughout the system. Any types of value can be used on backing stores as well as in main store. So directories, for example, play no special role, since procedure

values, files or indeed any values can be held anywhere on disc. In Flex the actual external modules used by a program are put in the text which is to be compiled. The meaning of the text is not dependent on the particular context at the time when it was compiled, but is self-contained. This relieves some significant problems of configuration control. Exactly the same principle is used in the network. The system holds the actual remote capabilities, not just their names, so changes in the association of names with values are irrelevant and again much confusion is avoided.

The mechanisms that are to be described attach a network meaning to normal constructions of programming languages, in particular to the procedure statement. It will be shown that the use of procedures in distant machines appears exactly the same as the use of local procedures. Changes do not have to be made in order to use a procedure from another machine even if it involves procedure parameters; the remote capability mechanisms allow the existing procedures to be used. This feature has proved of considerable practical use. Procedures devised for one Flex machine have been converted into network services by the simple means of providing remote capabilities for them. Their parameters, results, exceptions and break-in properties remained unchanged. Of course, procedures which would require a large bandwidth for communication may be impracticable on the network, but the mechanisms would permit them.

Flex supports processes as well as procedure values. Processes are necessary when true parallelism is needed, but if the purpose is only to create a context, procedures are simpler and more flexible. Procedures are used as the preferred mechanism in Flex. This certainly helps when we wish to convert an existing procedure into a network service, since programs written in conventional languages are likely to be in terms of procedures - it is not normal to use a sin or cos process in Pascal. It is also the current practice to define abstract types by the operations which can be performed on them, which again corresponds more naturally to the procedures that can act on them, rather than to processes.

The next section gives a simple example of the use of remote capabilities. This is not intended to be realistic, but to exhibit some of the features which are needed. The third section shows how the remote capabilities can be encoded and used so that remote and local calls are very similar. Section 4 gives further examples. The fifth section discusses the initialisation of the system and what capabilities must be present when the computer is initialised, and the sixth considers the lifetime of remote capabilities.

4

2   A Simple Example

The Flex type system provides basic types and ways of constructing compound types. Among the basic types are Void, which is represented by zero bits, and the types Int, Real, Bool and Char. The type of a structure (a record) is understood to be a cartesian product and is written as, for example, Int × Real × Char. The type of a procedure specifies the domain and range, for example, (Int × Real) → (Int × Char × Bool). Vectors of values have types like Vec(Char × Real). There are other kinds of type, but these will suffice for the following example.

Let us suppose that we have in computer A an integer variable, v, and that we want to give ways of altering the variable and reading its current value to computer B. The actual alteration and reading has to be done by A, but B has to cause it to happen and must supply values to put in the variable and must receive the results of reading it. We construct two procedures in A, assignvar, which takes an integer parameter and alters the variable, and readvar, which delivers an integer result which is the current value of the variable. Since Flex has true procedure values, these procedures are first-class values *and are independent of any context which might have set them up.*

$$\text{assignvar : Int} \rightarrow \text{Void}$$
$$\text{readvar : Void} \rightarrow \text{Int}$$

A procedure accesses values which are given to it when it is called, values which it creates while it is running, and values which were passed to it from its context. On Flex the latter values, its non-locals, are bound with the code, by means of an operation called close by Landin (Landin 1964), to form an object which is the procedure. If some of the non-locals are references, those references (that is, the appropriate capabilities) are included. Since the capabilities are also true values, this makes the procedure into a true value which is now independent of the context which set it up. The procedures assignvar and readvar do have a non-local reference, in fact the reference to v, so each of them has a capability for v in its non-locals. There may well be no other references to v anywhere else in the machine.

These procedures exist and run in A and alter A's variable, v. Now *we create in A two new remote capabilities, different from any* previous ones, which A can recognise as its own, and we associate them in A's store with the two local procedures. A can send these remote capabilities to B, though we shall postpone for the moment

the discussion of how it does so. B can then make a remote procedure call to A in which one of these, say the remote capability for assignvar, is the procedure to be called and the parameter of the procedure is the integer which is to be assigned to v. A will, on receiving this request, translate the remote capability that it receives into its own local procedure, the local assignvar, and apply this procedure to the given integer parameter, thus changing v. It then sends back the result, in this case Void, which acts as an acknowledgement to B. The process in B which issued the remote call has been held up waiting for this acknowledgement or an exception to be returned, and continues when it arrives. Clearly, in the case of readvar, the value sent back - the result of the procedure - is the required contents of the variable v.

Of course the interaction between A and B occurred over an ordinary network, so the information had to be coded into bytes for transmission. It is important that the capabilities should not be forgeable. However, if the sending and receiving of the packets of bytes, and their coding and uncoding into proper values can only be done by microcode, or by trusted system procedures, then we have a safe system. Once again, in Flex, the transmission of capabilities is done by microcode and this ensures that the individual words created as a result of the transfer are safe to use. We cannot confuse capabilities with non-capabilities. But the transfer of compound objects is done under the control of the type system, and though we can convince ourselves that the actual transfer itself is correct in terms of types, since this transfer program is written once for all, we only know that the types of the values were correctly stated if the compilers are correct.

The two procedures in A were proper values and had to be associated with remote capabilities which were sent out. It might be that the procedures are not used anywhere else in A, so the fact that the remote capabilities have been sent to B must be sufficient to keep the local procedures and the association alive and protected from A's garbage collector. Care must be taken if B finishes with them, or if B fails, that the procedures are not kept alive indefinitely by the interface alone.

We can build on this example to show one way in which procedures can be communicated. Since procedures are true values in Flex, new procedures can be created inside a procedure call and delivered as its result. We could therefore produce a procedure, genvar, in A, such that when it is called it creates a new integer variable and two procedures like assignvar and readvar to update and read the

6

variable, and delivers the two procedures. Its type is therefore

$$genvar : Void \rightarrow ((Int \rightarrow Void) \times (Void \rightarrow Int))$$

a procedure delivering two procedures with types like assignvar and readvar. Consider what happens if we associate genvar with a remote capability and send it to B. B can call the remote capability with a Void parameter, which activates genvar in A, creating a new variable in the store of A and two procedures which are local to A and have the capability for that variable bound into them. The capability for the variable occurs nowhere else in A. Then A has to send the two procedures back to B as the result of the remote call. This can be done by creating new remote capabilities for them and sending the capabilities to B. B is now in the position of the earlier example and can assign to the variable in A and read its value by using the two remote procedures to do so. Each time B uses the remote capability for genvar, A creates a new, different variable and two procedures bound to it, creates remote capabilities for these procedures and sends the two new remote capabilities back to B. All these interactions, and therefore in this sense the high-level protocols which are possible with genvar, are implicit in its type.

We can see from this example that the possession of a remote capability for a procedure of an appropriate type enables us to create and pass around new remote capabilities. Remote capabilities can be transmitted as procedure parameters as well as procedure results. In section 6 we consider the initial remote procedure which must be owned in order to start off the indefinite transmission of other procedures, and particularly consider its type.

Each of the local procedures could have been used locally in A. The only operation that was necessary to make this local "service" available over the network was to create and send out the remote capability for genvar. The creation and transmission of the other capabilities was automatically done by the system.

It would be possible to use remote capabilities standing for other types of value than procedures. We might, for example, have a remote reference. A remote reference would have to have remote assignment and remote dereference operations defined on it, so we would have such operations as well as remote procedure calls. Likewise remote arrays would have remote indexing operations. It seems unnecessary to introduce these types, since remote assignment and remote indexing would have to be implemented by means of some assignment code in the originating machine, in effect a remote procedure. The mechanisms which have been defined will suffice for the defining operations for all types of value, including new abstract data types

7

in the object-oriented sense. Indeed, the example above shows how
this would be done for references.

3 Encoding the remote capabilities

The Flex type system is intended to be sufficient for type
checking operating systems. It contains basic types, type generators
(including ones for backing store), new abstract types and new type
generators, and polymorphic types. In this it is like the type system
of ML (Gordon, Milner and Wadsworth 1979), with the addition of type
generators for the basic backing store types and, as we shall see,
for network types. Unlike ML, it permits programs which are created
by the compiler to inherit type checking, which will also be correct.
Part of the mechanism for this is the use of an infinite union type,
which is the union of all types. In Flex this type is called Moded. The
procedures which control the transmission of data are written to
operate on Moded values, so that, when they check types, that type
checking is guaranteed by the correctness of the compiler which
created the procedures.

We are trying to achieve a system in which the network is
transparent. Procedure calls, procedure parameters and results,
break-in to procedures and exception handling are all to be like those
for local procedures except that, by using a remote capability
instead of a capability for a local procedure, we are activating a
different computer. In order to issue a remote procedure call we
need a remote capability for the procedure that is to be called and
some parameters to give to it. From these we must form a packet by
encoding them into a byte form. The result will come back in the form
of a packet of bytes, which must be decoded into the proper form of
the procedure's results.

At the other, receiving, end a packet arrives containing the
encoded remote capability, which the receiver recognises as
corresponding to a particular local procedure. The rest of the packet
contains the encoded parameters, which must be decoded into proper
local values. These are passed to the actual procedure which is run in
a new process created for the purpose. The result of this local call
is then encoded and sent back to the originator. We shall not consider
the detail of how the network transfers the packet of data, though we
shall describe below how time-out, rejection of the request and
abortion of the procedure once it has started can be fitted into this
picture. It can be seen that the capability for the procedure to be
called is dealt with differently from the parameters and the results.

It would be possible to write the procedures, flatten and

8

unflatten, which encode and decode the parameter and result values, and give them the types

$$\text{flatten} : \text{Moded} \rightarrow \text{Vec Char}$$
$$\text{unflatten} : \text{Vec Char} \rightarrow \text{Moded}$$

If we did this we would have to encode the actual mode into the vector of characters in the packet. It would be possible to write flatten and unflatten this way, and have them known to be correct in terms of type because of the type checking inherited from the compiler which produced them. However, as we shall see, both ends of the transaction know the types that are being transmitted. Thus in order to avoid the waste of encoding the types into the packet but still maintain type-correct programming the procedures have been given the types

$$\text{flatten} : \text{Moded} \rightarrow \text{Vec Char}$$
$$\text{unflatten} : (\text{Vec Char} \times \text{Type}) \rightarrow \text{Moded}$$

The form of these is not symmetric. The flatten routine does not encode the type into the packet, but at the end of the transaction where unflatten is used the type of the data is always known, as we shall see below, and can be supplied as a parameter. Supplying the type in this way does not break the type checking rules, since even if this information is wrong it is still only possible to create legal data. If the wrong type were to be supplied, either the incompatibility of this with the packet would be discovered, or the result, ··· le being of the specified type, would not be the result that was required.

Such values as Int, Real, Char and Bool can easily be encoded, and so can structures, vectors and unions of them. We are assuming in this that the representations of the primitive values are the same in both the source and destination machine. The more complex considerations involved by different representations have not been studied in the implementation on Flex.

The more interesting question is how to encode the values which are procedures. Suppose that we are in the following situation. Computer A is about to issue a remote procedure call upon computer B, using a remote capability for a procedure in B, say m, which has a procedure parameter. We wish to arrange that the procedure which A is going to supply as the actual parameter, say f, is encoded and passed over to B. But B's procedure m, which is a normal procedure in B, is expecting an ordinary procedure parameter. We have therefore got to produce in B an ordinary procedure, say bf, to pass

9

to m. When bf is called it must issue a remote call to A, where the real procedure f resides, wait for the result, decode the result into an ordinary B value, and return this to m. The procedure bf must have its proper type in B.

Suppose then that the procedure f in machine A has type Par → Res and we are applying flatten to f in A. Flattening the procedure must produce a new remote capability to transmit to B, and it also creates, in machine A which does the flattening, an association between that capability and the procedure. However the procedure that is associated with the capability is a modified version of the original procedure f. In fact we create a new remote capability, cf say, in A, and a new procedure, ff say, in A

> ff : Vec Char → Vec Char
> ff = λv. flatten(to_moded(f(unflatten(v, Par))))

and associate ff with cf. The effect of ff is to unflatten the rest of the packet containing the remote call from B, producing a value of type Par. The local procedure f is applied to this, the result is converted to a Moded value by to_moded, and flattened back into a packet. Hence the procedure ff accepts packets, translates them into proper parameters for f, calls f and translates the results back into packets, which it delivers.

Note that we can keep ff in an association list for the remote capabilities because true procedure values are manipulable objects. As long as the association between cf and ff persists, we shall be keeping ff, and therefore also f, alive and protected from A's garbage collector. We now encode cf in a unique way, so that its encoding is different from that of every other kind of value and from any other remote capability created either in this machine or any other. To do this we need to incorporate something equivalent to the machine A's identity and the time of creation. We also include the type of f encoded as a sequence of bytes.

It is worth pointing out the level of protection provided by the assumed correctness of various parts of the system. If the microcode controlling the transfer of capabilities is correct, then we cannot confuse capabilities with non-capabilities, so we can be sure that we have only the capabilities that we are allowed to have and are using them in the proper way. But we may be wrong about the Flex types of the objects that have been transferred and indeed we may have transferred the wrong values, though they must be ones to which access was possible. If the type checking derived from the compiler is also correct, then we have data of the right type, but it might not be the data which we thought was the parameter of the remote call. If

the procedures flatten and unflatten are correct then we have the right data.

The remote capability cf is associated with ff in such a way that A can find ff if it is given cf. Let the procedure to find ff from cf be find_proc, where

find_proc : RemoteCap → (Vec Char → Vec Char)

so if A receives a remote call, in which the capability is cf and the packet representing the parameters is v, it can implement this by launching a process to obey

find_proc(c)(v)

and sending the bytes delivered by this back as the result of the remote call.

B needs to issue the call, and to do this the primitive procedure remote_call is defined.

remote_call : (RemoteCap × Vec Char) → Vec Char

B therefore obeys a procedure bf, which has the same type as the original procedure f. It flattens the actual parameter and uses the resulting Vec Char and the capability cf in a remote call. It then unflattens the Vec char which is returned, producing an object of the correct type. Obeying bf in B thus has the effect of calling f in A. Note that if f has procedure parameters, so will bf. These will be supplied in B and flattening them will create remote capabilities and associated procedu. s. So when f, running in A, calls its procedure parameter the call is transferred back to B and is obeyed there.

If an exception occurs in A during the execution of the procedure f, perhaps an overflow, an index out of bounds or a user stimulated exception, we would like to have this transferred back to B, the originator of the remote call. Accordingly it is necessary for the call of flatten in the body of ff to trap the exception and deliver a version of it coded as a vector of characters. Likewise, when B receives the packet containing the encoded exception, the unflatten operation causes the appropriate exception in B. A time-out in the remote call, if one is implemented, must cause a time-out exception to occur in B. Flex implements exceptions and exception traps.

In Flex, when a process is launched, a procedure is delivered which when called will abort (break in to) the process. Over the network we want to achieve the same effect. B, having started a remote call in A, now wants to abort it. There is therefore the

11

provision to send from B a further packet, an abort packet, belonging to the remote call, on receipt of which A will abort the process it launched to obey f, and return the exception value it receives as a result, duly encoded as above, back to B. The call to abort the transaction had to occur in a different process in B from the one which issued the remote call, since that process waits for the reply from A. It is necessary to specify what happens if the abort packet arrives in A after the completion of the call of f. Since any abort has to be issued from a different process, it is always possible that the process in question has terminated before the abort is acted upon (it is impossible to schedule with semaphores or monitors, in the very nature of an abort). So the correct programming of aborts always allows for the possibility that it is too late, and this rule can also be insisted upon for remote calls. Hence it is adequate for an abort package which arrives too late to be ignored.

It can be seen that the remote procedure call in B behaves exactly like an ordinary procedure call in B, in respect of parameters, results, exceptions and break_in. So it is often possible to test programs locally, not using the network, and then when they are working, to use them remotely. This can simplify the debugging process.

4   Further examples

4.1   Remote use of dictionaries

A dictionary is an abstract type with five defining procedures, one to make a new dictionary, one to add names and values to a dictionary, one to look up names in a dictionary, one to delete and one to produce a visible form of the dictionary.

> new_dict : Void → Dict
> add_dict : (Dict × Vec Char × Moded) → Void
> find_dict : (Dict × Vec Char) → Union(Moded, Void)
> delete_dict : (Dict × Vec Char) → Bool
> show_dict : Dict → EditableFile

Using the object-oriented approach we can say that each dictionary which is made up by new_dict consists of a group of four procedures, derived by binding in the actual dictionary. We can write a procedure, make_dictionary, which calls new_dict, and then delivers four

12

procedures.

make_dict : Void → ((Vec Char × Moded) → Void ×   {add}
                             Vec Char → Union(Moded, Void) ×   {find}
                             Vec Char → Bool   ×   {delete}
                             Void → EditableFile)   {show}

Note that a dictionary can contain values of any type and holds them
as Moded values. The look-up procedure, find_dict, produces a Moded
value if the identifier is present, and a Void value if not. The only
operations on the resulting Moded values are those which can be
produced by the compiler, so type checking has not been breached.

We can use a dictionary across the network by passing some or all
of the procedures delivered by make-dict in the form of remote
capabilities. Procedure values, and so the object oriented approach,
are being used across the network in exactly the same kind of way as
they are used locally.

4.2 Different file transfer protocols and the associated types

We consider how the type of the procedures characterises the
kinds of communication. Suppose that we are doing a simple file
transfer of named files from one machine A to B. Let A provide for B
the remote capability for

transfer1 : Vec Char → (Void → Vec Char)

in which the parameter gives the name of the file to be looked up in
some particular dictionary in A, and the result is a procedure for
producing successive lines of the file. The actual transfer1 runs in
A, having been remotely called from B. It creates a procedure to
deliver the lines, flattens it giving a remote capability, and sends
that remote capability back to B. B now issues a remote call using
this capability each time it wants a new line from A. So it is B which
is the active partner in the actual data transfer and determines when
each line is to be sent. On the other hand suppose that A provides

transfer2 : (Vec Char × (Vec Char → Void)) → Void

The first parameter is again the name of the file and the second is a
procedure for receiving successive lines of the file. Once again
transfer2 is local in A and remotely called from B. But this time B has
to create, flatten and send a remote capability for a procedure local

13

to it, which will deal with the lines when A cares to produce them.
This time it is A which controls the actual data transfer. So in the
same way as in a purely local type scheme, we can see the kinds of
interaction which are possible across the network, just by examining
the types of the procedures involved.

4.3 A command line interpreter

A command-line interpreter could have the type

interpret : ((Void → Vec Char) × (Vec Char → Void)) → Void

The first parameter delivers characters to be interpreted each time
it is called and the second parameter receives messages to display in
return. If the local procedure for interpret lies in A and B has a
remote capability for it, then B can activate the command interpreter
in A, passing procedures which A will call in B to produce the lines to
interpret and display the messages. If A fails when B is accessing it,
B will get a time-out exception, which can be trapped and turned into
an appropriate message. Note again that the type of interpret is such
that it could be used locally in A, and so tested in A alone before
being used on the network. We may expect this, since an object-
oriented procedural approach will of its nature lead to a procedural
definition which can be used remotely. This will usually be possible,
but not always useful, since the bandwidth of local communication is
likely to be much higher than the bandwidth available from the
network. It is not likely, for example, that large block moves of
pixels could usefully be performed if the data has to go across the
network. Hence remote graphics programs are likely to have to
communicate in a coded way, whereas within one machine this need not
be so.

5 Initialisation

All the kinds of transfer which have been discussed so far have
needed the possession of a remote capability before anything can
happen. Given an appropriate capability more capabilities can be
propagated around the system. Indeed given a suitable capability
involving the type Moded, we can arrange to transfer a capability of
any type. But we do need at least one capability to start all this off.
This has to take the form of some capability which a computer
automatically has as soon as it is activated.
On the Flex system we have chosen to provide an initial function

14

first_fn

first_fn : ComputerIdent → (Vec Char → Moded)

which is given the identification for a computer and delivers a
function from names to Moded values. The names are looked up in a
particular dictionary on the nominated computer.

6 Life-times

When machine A flattens a procedure in order to send a remote
version of it to a distant computer, the procedure is kept alive and
protected from the garbage collector by being associated with the
corresponding remote capability in the list used by find_proc. The
distant procedure can in turn pass the remote capability for this
procedure on to other computers, which could use it directly to call
the procedure in A. If A's store is not to become choked up by these
procedures it must garbage collect them away, and to do so it must
discover whether any of the other computers which are still on the
network are keeping the remote capability alive in its store and might
therefore use it. Computer A therefore periodically asks all the
other computers about each of its own remote capabilities. If none of
the others requires the capability, then A can remove it from the
association list, and the procedure will be removed at the next local
garbage collection. The other computers therefore have to have
access to all the remote capabilities which they hold, in order to
answer this question.

The identification of the computer which is included in the
encoding of remote capabilities, serves two purposes. It enables the
system procedures to send the information to the correct place to be
acted upon, and it is part of the unique identification of the remote
capabilities. The remote capabilities that we have described only
have a meaning as long as the computer in which they originated is
switched on. So the network address is a sufficient identification of
the computer. In systems where the computer moves about in the
network, cellular systems, the address which is used to identify the
computer will serve.

When a computer is switched on again, after having been off for a
period, it is necessary to avoid confusing packets created in the new
incarnation with old packets still in the network. A time of switch-on
is therefore also a necessary part of the identification of a
computer.

15

7 Conclusions

We have described mechanisms for interactions between machines on a network which preserve the safety of capability machines and include a network-wide type checking mechanism with inheritable checking and abstract data types. This system has been implemented on a network of Flex machines and is in use.

The already existing system of capabilities was clearly essential as a base for remote capabilities. However, it was the presence of true procedure values that made the rest of the system and its implementation relatively easy. It would not have been possible to use this method if procedure values could not have been kept in an association list, regardless of their origin in particular programs.

In the exploration of the system it has become apparent that we need a concept of "universal" capabilities, that is capabilities which have the same purpose network-wide. For example, a Pascal compiler in the local filestore of machine A is related to the Pascal compiler in machine B, even if they are not at the same level in terms of updates. We may want to send a procedure call to a machine saying "use your Pascal compiler" rather than "use this specific remote capability". Another example lies in the use of new abstract data types. It is natural to implement these as remote capabilities, but once again we want the meaning of types to be the same across the whole network. We want to refer to a particular abstract data type, using the same "universal" capability in a transaction with any machine on the network, not a different capability for each machine which happens to stand for the abstract type in that machine. The required properties of these universal capabilities are being explored in the Flex system.

The implementation has not addressed the question of mixed types of machine. Clearly, data may have different forms which must be translated in transactions between different types of machine. There are further, more difficult problems. The Pascal compiler, though still recognisably serving the same purpose on two machines, does not produce the same form of code for different machines. Some uses of a capability for the compiler will mean "use your compiler", others, such as updates, will mean "use the type A compiler", still other will refer to a specific capability.

So far all Flex implementations have been microcoded. The system-wide types would present no problem on a conventional computer, and the procedure values present only the problem of achieving an efficient implementation. But the capabilities present a more serious problem. One approach would be to depend entirely on

16

the type system and the checking provided by the various compilers in the system. The reliability of compilers is not at present good enough to make this a satisfactory solution, and we would like to allow for the possibility of the development of new compilers which will certainly go through a stage of containing errors which must not be allowed to crash the whole network.

Accordingly an abstract machine has been developed, known as Ten15 and based on algebraic principles. This machine is implementable on conventional computers and uses the Flex type system, enforcing it with complete rigour. We therefore have to depend for our network correctness only on the correctness of the implementation of Ten15 which can be used as target by many compilers. The Flex Ada and Pascal compilers already use Ten15 as their basis.

References

I.F.Currie, J.M.Foster
  "Curt : the command interpreter language for Flex"
  RSRE Memorandum 3522, Sept 1982

I.F.Currie, P.W.Edwards, J.M.Foster
  "Flex firmware"
  RSRE Report 81009, Sept 1981

D.M.England
  "Capability concept mechanisms and structure in System 250"
  Rev. Fr. Autom. Inf. Rech. Oper. (France)
  Vol 9, pp 47 - 62, Sept 1975

R.S.Fabry
  " Capability based addressing"
  Comm. ACM Vol 19, pp403 - 412, July 1974

J.M.Foster, I.F.Currie, P.W.Edwards
  "Flex: a working computer with an architecture based on
  procedure values"
  Proc. International Workshop on high-level architecture, Fort
  Lauderdale, Florida (Dec 1982) pp 181-185

J.M.Foster, C.I.Moir, I.F.Currie, J.A.McDermid, P.W.Edwards,
J.D.Morison, C.H.Pygott
  "An introduction to the Flex computer system"
  RSRE Report 79016, Oct 1979

M.J.C.Gordon, A.J.Milner, C.P.Wadsworth
  "Edinburgh LCF"
  Springer-Verlag 1979

P.J.Landin
  "The mechanical evaluation of expressions"
  Computer Journal Vol 6 No 4 pp 308 - 320 Jan 1964

B.Liskov, R.Scheifler
  "Guardians and actions: linguistic support for distributed
  programming"
  ACM Trans. Prog. Lang. and systems. Vol 5, pp 381 - 404 July 1983

R.M.Needham, R.D.H Walker
  "The Cambridge CAP computer and its protection system"
  Operating System Reviews
  Vol 11, No 5, pp 1 - 10, Nov 1977

A.Z.Spector
  "Performing remote operations efficiently on a local computer
  network"
  Comm. ACM Vol 25, No 1, pp 39 - 59, April 1982

J.E.White
  "A high-level framework for network-based resource sharing"
  AFIPS Conf. Proceedings, National Computer Conf. Vol 45, p561-570
  1976

Xerox Corporation
  "Courier: the remote procedure call protocol"
  Xerox Report XSIS 038112, Dec. 1981

DOCUMENT CONTROL SHEET

Overall security classification of sheet ........UNCLASSIFIED.. ....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter
classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Memorandum 3947	3. Agency Reference	4. Report Security Classification
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location Royal Signals and Radar Establishment		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title
REMOTE CAPABILITIES IN COMPUTER NETWORKS

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers)   Title, place and date of conference

8. Author 1 Surname, initials FOSTER, J M	9(a) Author 2 CURRIE, I F	9(b) Authors 3,4...	10. Date	pp. ref.
11. Contract Number	12. Period	13. Project	14. Other Reference	

15. Distribution statement UNLIMITED

Descriptors (or keywords)
continue on separate piece of paper

Abstract
 System-wide types, capabilities and procedure values can be used in computer
 networks to improve control of access and flexibility. This paper describes
 a network system based on these ideas. They provide a standard, uniform way
 of referring to data and using procedures in remote computers, with control
 over access and system type-checking. A working experimental version of the
 system exists, implemented on Flex computers, which provide microcoded support
 of capabilities. The mechanisms are such that normal programs written in terms
 of procedures, even those using procedure parameters, can be converted into
 network services without change. A possible way of implementing a similar
 scheme on conventional hardware is discussed.

S80/48

END

DATE
FILMED
2-87